PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 43, NO. 1, PP. 3–17 (1999)

APPLICATION LAYER ANYCAST

Miklós BERZSENYI*, István VAJK* and Hui ZHANG**

*Department of Automation
Budapest University of Technology and Economics
H–1521 Budapest, Hungary
e-mail: bermik@sun.aut.bme.hu, vajk@aut.bme.hu
**School of Computer Science
Carnegie Mellon University
Pittsburgh 15213, USA
e-mail: hzhang@cs.cmu.edu

Received: Oct. 1, 1999

Abstract

In this paper, we present a new approach to application layer anycasting. The key to anycast is making it possible for clients to efficiently find the 'best' server for a given application in an unknown group of servers. The anycast service makes a wide range of new multimedia applications possible, and will be part of future integrated services networks. We designed a selective anycast protocol, which is aimed at picking the right server based on application specific metrics, such as network delay and server load. This paper considers server-choosing metrics and efficient mechanisms to compute these metrics. We also present simulation results, which show our approach's merit, and proves that anycast can significantly improve the performance as compared to the traditional methods.

Keywords: anycast, service enabling platforms, integrated services packet switched networks.

1. Introduction

In the Internet today, the World Wide Web is becoming an increasingly important application. Further, some of the most visited web sites consist of uncacheable data, either because the data is private, the data becomes stale quickly, or the data is customized for each user. We have also observed that network service has been degrading for the last few years, until currently, clients are able to retrieve only hundreds of bytes per second during the Internet's busiest periods.

The traditional method to overcome scarce network resources has been caching [3]. But caching becomes less and less effective as servers with uncacheable data become more common. Since only servers are able to deliver data to clients, we must consider server–initiated mechanisms for improving performance. Previously, server–initiated solutions have consisted mainly in infrastructure improvements such as using more powerful computers or buying faster network connections. While these solutions are certainly effective, they are not perfect. A congested link inside the network is the great equalizer among server hardware: regardless of how fast the server can push data into the network, the client will only receive it as fast as the congested link will allow.

In an attempt to avoid the congested link problem, server maintainers have begun to establish 'mirrors' of a server at other locations in the network. If a client is dissatisfied with the performance of a server, it may switch to any of the mirrors. However, unless the client is clairvoyant, it will have difficulty picking the server that can supply it with the best performance. Still, as anyone who has ever beaten the odds and picked the right mirror will attest, this approach has merit. By scattering servers around the Internet, there is less of a chance that every path between a client and the servers will be congested. The missing piece of the puzzle is how to choose automatically the best server for a given client.

In 1993, PARTRIDGE et al. [9] introduced the concept of anycast, a network service that would deliver a message to anyone of a group of servers in the Internet. A selective anycast service is exactly what is needed to choose the best mirror server. By 'selective,' we mean that the choice of server should be based on some application-specific metrics. This paper will consider server-choosing metrics and efficient mechanisms to compute those metrics. We believe that the anycast service is a new building block for future integrated services networks as it provides new capabilities. The rest of the paper will be structured as follows: Section 2 will discuss previous work in this area. Section 3 will discuss the design space for anycast protocols. Section 4 will cover simulation experiments which explore the tradeoffs among our metrics. Finally, Section 5 will put our work in perspective.

2. Related Work

There have been several works dealing with network probing to choose a server from a group. GUYTON et al. [6] studied a number of mechanisms ranging from monitoring routing updates to using a series of statically-located hosts which probe the network. In Guyton's terminology, our work focuses on 'reactive' mechanisms while his work focused on 'proactive' mechanisms (though not to the exclusion of reactive mechanisms). In other words, Guyton's emphasis was on server-initiated network information collection, while our work emphasizes gathering information about a client from the client's query messages to the anycast group.

As we have said, anycast was first proposed by PARTRIDGE et al. [9]. They envisioned a network layer service which would route a client's datagrams to the server that is the fewest number of hops away. LEVINE et al. [7] also developed a network layer method of finding the closest host in a group. This version of anycast is not well-suited to the problem we have chosen to solve because of its inflexible metrics. We believe that server choice should be based on a variety of metrics including, but not limited to, hop count. Some servers may wish to use server load, or achievable throughput between the client and server. Also, we believe that the metrics information, given that it is server-specific, is best left out of the network layer. BHATTACHARJEE et al. [1] as well as FRANCIS [4] have considered the infrastructure necessary to support application-layer anycast.

Bhattacharjee's scheme involves altering clients' Domain Name Server (DNS)

resolvers to choose one server from a group when presented with an anycast address. The choice of server can be made according to any server-defined metrics which fits inside their rather general framework. They also explore several metrics for choosing servers. Our work is meant to coexist with their work. Our mechanisms for finding anycast servers could be implemented inside the DNS framework.

Finally, a company named Bright Tiger [2] has a product called Cluster CATS which claims to transfer users to the appropriate web server based on network congestion and server load, among other things. Of course, Bright Tiger's approach is proprietary information, while our work will be released into the public domain. Bright Tiger's web pages contain some information about their system. It requires no changes to clients, so client-initiated network probing is out of the question. It seems to be based on estimating the network latency between the client and server which is then used as a combined network distance and bandwidth metrics. Unfortunately, this implies that a client must connect to a server and transfer some data before the server can redirect the client to a better server. Since the server has no information about the client's links to other servers, it needs to guess which other server would better suit the client. The result is that a client may be ping-ponged from server to server if the network is heavily congested.

Other somewhat less related work includes the Internet Engineering Task Force's (IETF) Service Location Protocol [5], a way to find a server given only a text name of the service that it implements. This protocol is focused on returning the address of any server while our work aims to return the address of the best server for a client. Also, the service location protocol is not designed to scale beyond a large intranet. Finally, we consider caching [3]. While caching has the same goal as our work, it takes the opposite approach. With caching, the first client to access a document forces the document to be cached so subsequent clients can take advantage of the nearby cached copy. Our work focuses on giving all clients quick access. In addition, as has already been mentioned, many services are uncacheable. In general, though, caching and anycast should complement each other.

3. Anycast Proposal

The key to anycast is making it possible for clients to efficiently find the 'best' server in an unknown group of servers. When designing an anycast protocol, there are a number of different metrics that might contribute to the determination of which server is 'best'. These include: (a) the network delay between the client and each server, (b) the available bandwidth between the client and each server, (c) whether or not the request can be served out of the cache at each server, and (d) the load at each server. For example, the importance of (a) varies depending on the amount of processing each request will require, the importance of (b) varies depending on the amount of data we expect to transmit, the importance of (c) varies depending on the degree to which caching is beneficial and plausible for the application in question, and the importance of (d) varies depending on the server processing required by the

request. Assuming that the amount of data transmitted is small (so that clients are indifferent to available bandwidth), and that caching behavior is either irrelevant or will be good regardless of the distribution of requests, we focus on (a) and (d); that is, we are primarily concerned with request latency, and we assume that latency mainly depends on network delay and server load, plus any overhead imposed by the anycast protocol.

3.1. Proposal Overview

Since clients are initially unaware of the composition of the server group, they must be able to obtain the identity of one or more members by sending a request to some well-known authority. We call this request a FindServer request. The authority can be a single server contacted via ordinary unicast, but relying on a single server suffers from all of the well-known problems with centralized systems. A more scalable and fault-tolerant solution is for this authority to be a set of servers which can be contacted via multicast. Which of the anycast servers should subscribe to this multicast group? Clearly, having every anycast server to receive every FindServer request would introduce unacceptable overhead. There are two ways to limit the number of anycast servers which receive a FindServer request: we can use Time-To-Live (TTL) scoping to limit the range of a FindServer request, or the multicast group can include only a subset of the anycast servers.

Once the multicast servers receive a FindServer request, how do they ensure that the client is directed to the best server, i.e. the server which is least loaded and with whom it experiences the lowest network delay? If we are using TTL scoping, then each receiving server can simply respond with a local prediction of load (using some agreed upon metrics), and the client will be able to determine relative loads and delays. If we are multicasting to a subset of the servers, then the members of the subset must obtain load information from the servers which are not in the subset. For simplicity and efficiency, this implies some hierarchical organization of the anycast servers into regular servers and representative servers, along with a grouping of the servers into clusters where each cluster has exactly one representative. Then, each regular server will report load information to its representative so that the representatives can respond to FindServer requests with the identity and load prediction of the predicted least-loaded server in their cluster. Notice that the delay that the client calculates will reflect its distance to the representative instead of its distance to the server identified in the response. To minimize this distortion, clustering needs to be based on delay between servers.

In the following two sections, we describe the basic TTL scoping scheme and the basic clustering scheme in greater detail. For each scheme, we present the protocols by which new servers are added to the server group (the Join protocol), the protocol with which clients obtain server identities and choose a server to receive their data request (the Anycast protocol), and the protocol by which the server group recovers from network and server failures (the Recovery protocol).

In subsequent sections, we compare both schemes, identify their weaknesses, and propose improvements.

3.2. Basic TTL Scoping Scheme

All servers subscribe to a multicast group, the AllServers group, and each server keeps track of its predicted future load based on its current load and the number of FindServer requests it has responded to in the last time period.
Join protocol. New servers simply subscribe to the AllServers group.
Anycast protocol. When using TTL scoping to limit the number of servers that receive a FindServer request, clients need to discover an appropriate TTL value; that is, clients must perform an expanding ring search (ERS). Servers which receive FindServer requests respond with their load prediction. When a client receives a response, if the load prediction is below some threshold, then it sends its data request to that server. Otherwise, it waits until it either receives a response which contains a sufficiently low load prediction, or the allowed waiting period has passed. If the allowed waiting period has passed and a minimum number of responses were received, then it sends its data request to one of the servers, selected based on response time and load prediction. Otherwise, it continues the ERS. The client caches its final TTL value. The server which receives the data request processes it and sends the appropriate response.
Recovery protocol. No recovery protocol is necessary.

3.3. Basic Clustering Scheme

All servers subscribe to a multicast group, the AllServers group, and representative servers also subscribe to another multicast group, the Representatives group. Each server keeps track of the identity of its representative and sends it a load information update once per time period. Each representative acknowledges its load information updates, combining them with information it maintains on the number of times it recommended each cluster member in the past period to predict each cluster member's load in the near future. Representatives also update their cluster membership information, if necessary.

Join protocol. (*Fig. 1*). As mentioned in the overview, clusters have to be created carefully so that the delay calculated by a client, which is the delay between itself and each representative server, is a fairly accurate portrayal of the delay between the client and the server suggested in the representative's response. Therefore, new servers have to gather delay information before they can decide on which cluster to join. A new server multicasts to the AllServers group and uses the responses to build a list of the servers (this list need not be complete). It then measures the Round Trip Time (RTT) between itself and every other server. If it discovers that the RTT between itself and at least one representative server is below some threshold,

then it sends an initial load information update to the closest of these in order to join an existing cluster. Otherwise, if it discovers that the RTT between itself and at least one regular server is below the threshhold, then it asks the other server to become a representative. The other server will promote itself by multicasting to the AllServers group, after which each regular member of its original cluster (and any other cluster) may choose to join the new cluster, and the new server will join the new cluster. Otherwise, the new server becomes its own representative (i.e. a one-member cluster).

```
New Server:
      QueryOtherServers(Allservers, Multicast)

Upon Receiving answer from any server:
      if (MeasureRTT(me, Server)< Treshold) then
           Remember (Server)

At the end of the RTT measurement
      if (ChooseClosestRepresentative is_not_empty) then
           JoinCluster(RepresentativeServer)
           End
      if (ChooseClosestServer is_not_empty) then
           AskToBecomeRepresentative(Server)
           JoinCluster(RepresentativeServer)
           End
      AskToBecomeRepresentative(Me)
      End
```

Fig. 1. Join protocol

Anycast protocol. (*Fig. 2*). A client multicasts a FindServer request to the Representatives group. Representative servers respond with the identity and load prediction of their predicted least-loaded cluster member. When the client receives a response, if the load prediction is below some threshold, then it sends its data request to that server. Otherwise, it waits until it either receives a response which contains a sufficiently low load prediction, or the allowed waiting period has passed. If the allowed waiting period has passed, then it sends its data request to one of the servers, selected based on response time and load prediction. The client caches at least the identity of the representative server whose recommendation it used. The server which receives the data request processes it and sends the appropriate response.
Recovery protocol. (*Fig. 3.*) If a regular member does not receive an acknowledgment for one of its load information updates within a time period, it sends another. If this second update also times out, then it multicasts to the AllServers group that its representative is unresponsive. The regular servers who belonged to this cluster exchange messages to determine their RTTs to each other, and then each advertises the average of its RTTs. The server with the smallest average promotes itself by

```
Client:
      FindServer(ReprentativeServers, Multicast)

RepresentativeServer:
      SendLoadInformation(LeastLoadedServer, Client)

Client upon receiving answer:
      if (ServerLoad<treshold) then
            SendDataRequest(Server)
            Cache(RepresentativeServer)
            End
      Else StoreLoadinformation(Server)

Client upon timeout:
      ChooseBestServer(RTT, LoadInformation)
      SendDataRequest(Server)
      Cache(RepresentativeServer)
      End
```

Fig. 2. Anycast protocol

multicasting to the AllServers group, and the other servers either update their choice of representative or (if their RTT to the new representative is above the threshold) run the join protocol. If a representative does not receive a load information update from one of its cluster members within a time period, it explicitly requests this information from that member. If there is no response to this request, then it assumes that the member no longer exists and updates its cluster membership information.

3.4. TTL Scoping versus Clustering

A comparison There are four drawbacks to the TTL scoping scheme. First, performing an ERS adds latency. Second, hop count may not correspond to network delay, so that the set of servers that receive the FindServer request may not include the set of closest servers. Third, the network topology and distribution of servers may be unfriendly to ERS; that is, the percentage of servers reached may always be either too small or too large. And, finally and most importantly, the servers will not be load balanced if the distribution of clients does not match the distribution of servers. This imbalance could get quite extreme. For example, if clients generally need to cross a backbone link to reach a server, but the servers are at vastly different numbers of hops from the backbone, then servers which are far from the backbone will never receive any requests. These drawbacks prevent the TTL scoping scheme from being a good, general solution.

In contrast, using clustering takes approximately the minimum additional

```
Regular Server:
      SendLoadInformation(RepresentativeServer, cyclic)
      If (Timeout(Acknowledgement)) then
            SendLoadInformation(RepresentativeServer, 1)
            If (Timeout(Acknowledgement)) then
            RepresentativeUnavailable(Representative,
            AllServers, Multicast)

Each Regular Server, which belonged to the same cluster
      for (every server in the cluster)
            MeasureRTT(me, Server)
      AnnounceAverageRTT(cluster)
      The server with the smallest average
      AskToBecomeRepresentative(Me)
      End

Other servers
      Run Join Protocol

Representative server
      If (Timeout(LoadInformation(Server) then
            AskLoadInformation(Server)
      If (Timeout(LoadInformation(Server) then
            RemoveFromClusterList(Server)
```

Fig. 3. Recovery protocol

latency for application-level anycast (i.e. one additional RTT for the FindServer request and response), is not dependent on hop count accurately reflecting network delay, is flexible with respect to network topology and server distribution, and allows some degree of load balancing across all of the servers. On the other hand, the server structure inherent in clustering imposes some additional complexity and overhead, requiring protocols to create and join clusters, to exchange load information between the regular members and the representative member in each cluster, and to maintain clustering in the face of network and server failures. In addition, there is no way to efficiently provide the accurate delay information obtained in the TTL scoping scheme, so clients must make their server choice based on less accurate information. Decreasing cluster sizes would increase the accuracy of the delay calculation, but it would also increase the percentage of servers which receive each FindServer request. This tradeoff between FindServer overhead and a client's ability to identify the best server is the greatest weakness of the basic clustering scheme.

3.5. The Modified Schemes

The main weakness of the TTL scoping scheme is its inability to automatically balance load among the servers if the distribution of servers and clients is not close to ideal. The problem is that, if a client increases its TTL in order to reach more servers (so that load can be more evenly distributed among the servers), then the FindServer request overhead will also increase. We can allow clients to reach more servers without adding to FindServer overhead by combining TTL scoping with clustering. This modified scheme, which we call clustered TTL scoping, is exactly like the basic clustering scheme described in section 3.3 except that clients use ERS to locate representative servers (*Fig. 4*), and clients only need to cache a good TTL value.

```
Client:
      DoERS(TTL0, TTL256)
      Cache(GoodTTLValue)

DoERS(TTLstart, TTLend)
      FindServer(RepresentativeServers, Multicast, TTL)

Client upon receiving answer:
      if (ServerLoad<treshold) then
            SendDataRequest(Server)
            End

Client upon timeout:
      DoERS(TTLstart*2, TTLend)

RepresentativeServer:
      SendLoadInformation(LeastLoadedServer, Client)
```

Fig. 4. Modified join protocol using ERS

Although clustered TTL scoping no longer provides accurate delay information, and requires the additional complexity and overhead of clustering, it improves upon basic TTL scoping in two ways. First, as mentioned above, it has better load balancing properties because TTLs can be set much higher (reaching, through their representatives, a much larger number of servers) without increasing FindServer overhead. Second, it does not rely on having an ERS-friendly network topology and server distribution because reaching a large percentage of the representative servers simply causes this scheme to behave like the basic clustering scheme. If the topology and server distribution is ERS-friendly, then clustered TTL scoping also partially alleviates the problem with the basic clustering scheme; that is, clustered TTL scoping decreases the FindServer overhead for a given cluster size.

Another way to address the problem with the basic clustering scheme is to

permit representative servers to quickly decide whether or not to ignore a FindServer request, where the cost of making this decision and the cost of discarding a request should be much less than the cost of servicing the request. Assuming that the representative servers are able to correctly identify and discard a significant percentage of FindServer requests, then this modified scheme, which we call clustering with discard, will decrease the FindServer overhead for a given cluster size. Note that permitting discards requires a mechanism for disallowing discards in order to ensure that all clients will eventually be able to obtain the identity of a server. This can be simply accomplished by adding a flag in the FindServer request, which should only set the flag if they are extremely concerned with latency or were unable to obtain service without the flag.

```
Representative server upon receiving request
      If (hop_count<HopCountTreshold) then
            Decrease(HopCountTreshold)
            ServiceRequest
      else if (PredictsIdleServer) then
            Decrease(HopCountTreshold)
            ServiceRequest
      else
            Increase(HopCountTreshold)
```

Fig. 5. Clustering with discard

In clustering with discard (*Fig. 5*), there are three factors which could be used in the discard decision: the hop count from client to representative, past calculations of the relative load of the cluster, and the predicted load of the predicted least-loaded cluster member. The first two can be combined by using a hop count threshold which varies depending on the relative load experienced by a cluster. That is, a representative can keep track of what percentage of the data requests are being serviced by its cluster (calculated from information passed in the load information updates and by keeping track of the number of FindServer requests it received) and raise (or lower) its hop count threshold if the number of requests its cluster has serviced is below (or above) some low (or high) watermark of what constitutes its fair share of requests (based on what percentage of the servers are in its cluster). Upon receiving a FindServer request, the representative will discard it if the hop count is above its threshold and it predicts no idle cluster members. In other words, clustering with discard is like clustered TTL scoping except that the hop count is measured around the server instead of the client. The disadvantage of clustering with discard is that all representatives still receive all FindServer requests. The advantages are that it addresses the cluster size versus FindServer overhead issue even with ERS-unfriendly networks, it has no ERS overhead, and it has even better load balancing properties.

4. Evaluation

To evaluate and compare the performance of clustered TTL scoping and clustering with discard, we implemented both protocols as well as a base case in a packet-level simulator. The base case is to have a single well-known server, i.e. clients do not need to send FindServer requests.

4.1. Simulation Environment

We use a discrete event packet-level simulator which supports unicast and multicast routing, and UDP and TCP communication. Routing minimizes the number of hops/packet. Links are characterized by minimum delay, d, and loss rate. UDP communication is simulated by forwarding undropped packets hop by hop with a delay uniformly distributed between d and 2d. TCP communication is simulated by calculating end-to-end delay as the sum of ave path delay and bytes transmitted/bandwidth, where ave path delay is calculated by summing the average link delays of the links on the route, and bandwidth is calculated by plugging ave path delay and the average loss rate of the links on the route into equation 3 from [8]. Thus, unlike the real world, loss and delay in the simulation are independent of the number of packets traversing a link (so, for example, the single server base case does not suffer from network congestion).

All of our experiments use a single, randomly generated network topology (*Fig. 6*). The topology has two levels: a top level backbone network and a second level of regional networks. Each router in the backbone is connected to a regional router as well as being randomly connected to an average of four other backbone routers. Each regional network (of which there were 20) is connected to the top level backbone through a single router, and each regional router was randomly connected to an average of three other routers. Each backbone link was assigned a value for d of around 20 ms to 40 ms, while each regional link was assigned a value for d of around 1 ms to 5 ms. All links were assigned loss probabilities in the 0.1regional router is connected to one of the 400 hosts, and there are no multi-homed hosts. In our simulations, all hosts are either anycast clients or anycast servers. The servers are randomly distributed.

For the anycast simulations, we ran 10 server and 25 server tests. In the base case simulations, the single server was simulated as a 10 (or 25) processor machine; that is, each individual request took the same amount of time to process as in the anycast tests, but the single server could simultaneously process 10 (or 25) requests. In addition, the single server performs perfect load balancing amongst its processors. For simplicity, it was assumed that servers would process all non-data requests (e.g. FindServer requests) immediately and at no cost. This favors the anycast protocols. To partially account for the FindServer overhead in the anycast protocols, representative servers doubled their own load calculation when choosing a server to recommend in a FindServer response. Finally, servers were clustered

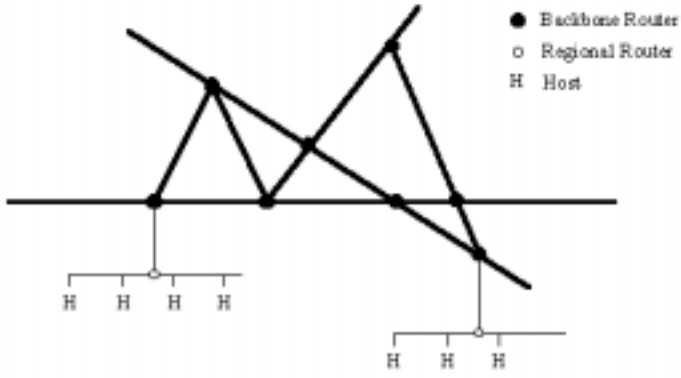

Fig. 6. Part of a possible network topology

with an RTT threshold of 100 ms. For the 25 server case, this resulted in an average of 4 servers/cluster. In the 10 server case, however, most of the clusters only contained a single member (there were 6 clusters).

4.2. Workload

The simulation was driven by a web server log. The log records 1,160,399 requests to the CMU CS web server from Nov. 23, 1997 to Nov. 30, 1997. For each request, the log contains a timestamp, a client identification, and the amount of data requested. Since the log records requests from 146,829 clients, each client in the simulation actually corresponds to many clients from the log. To avoid distorting client cache performance (i.e. a client's cache of an anycast server address), simulation clients maintain separate caches for each log client they represent. Server processing time for data requests is assumed to be linear with the amount of data requested.

4.3. Performance Metrics

We primarily evaluate each technique based on perceived performance and server overhead. The perceived performance is measured as client response time, which is subdivided into FindServer response time and data request response time. Server overhead is measured as the ratio of FindServer requests processed to the number of data requests sent; that is, both of our anycast proposals may result in multiple servers processing each FindServer request, so we measure the degree of redundancy.

In order to gain a better understanding of the performance of our protocols, we also measure the service time of data requests at the servers. This allows us to

determine how much of the client's perceived request latency was due to network latency. Also, by comparing the service times of the anycast protocols to that of the base case, we can determine how successfully each scheme load-balanced the servers.

4.4. Results

In *Table 1*, we see a breakdown of average client waiting time for each of the protocols in each of the topologies. The columns are marked as follows: 'ERS' corresponds to clustered TTL scoping, 'CWD' corresponds to clustering with discard, and 'Base' corresponds to the single-server base case. The first row, marked 'Find,' displays the average time to find a server to satisfy each request. The row marked 'Fetch,' displays the average time to load the data. The row marked 'Total' is the sum of the two rows above. All times are in milliseconds. Note that the base case has no find overhead since each client only goes to one server each time.

Table 1. Average Time (ms) for a client to receive data from the server

	10 servers			25 servers		
	ERS	CWD	Base	ERS	CWD	Base
Find	38.76	35.31	0	28.25	30.83	0
Fetch	679.69	672.56	659.71	747.93	745.67	741.67
Total	718.45	707.87	659.71	776.18	775.50	741.67

There are several interesting details in *Table 1*. We see that the time required to find a suitable server is 5% of the total time of each transfer for both ERS and CWD. The performance of the anycast protocols improves as the number of servers increases. This is also unsurprising as the average distance between a client and the closest server will decrease as the number of servers increases, and the same request stream should be more easily handled by a larger number of servers if the request stream is heavy enough to load the servers.

In *Table 2*, we see measurements of the service time as perceived by the server, and the degree of redundancy in processing FindServer requests. The columns are the same as in the previous table. The row marked 'Fetch,' displays the average time (in milliseconds) to process a request at the server. The row marked 'Find Redundancy' is the average number of servers that process each FindServer request.

Comparing the server Fetch values to the client Fetch values, we can see that the majority of client-perceived latency for the base case was due to network latency. In contrast, the majority of the client-perceived latency of the anycast protocols is due to server processing time. Comparing the average times of the ERS and CWD protocols, we see that, as expected, CWD is more successful at load balancing. We also see that, while adding more servers does decrease the average service

Table 2. Average Time (ms) for a server to service a data request, and average number of servers that process each FindServer request

	10 servers			25 servers		
	ERS	CWD	Base	ERS	CWD	Base
Find	375.62	321.34	248.63	320.27	301.83	242.67
Total	3.97	2.73	n/a	3.93	2.93	n/a

time slightly, 2.5 times as many servers translates into a relatively small decrease in service time, even after factoring out the minimum time (as measured by the perfectly load-balanced base case). Finally, by comparing the network latencies experienced in each protocol, we see that, as expected, the anycast protocols were able to reduce network latency by directing client requests to nearby servers. We also see that ERS was slightly more successful at reducing network latency than CWD. This is due to ERS's strict adherence to hop count from the client. In CWD, a nearby cluster may choose to ignore a request because it is heavily loaded. Since the penalty of bad load balancing is significantly more severe than the penalty of being serviced by a slightly more distant cluster, this appears to be a good tradeoff. Finally, notice that the number of servers that process each FindServer request is fairly low, indicating that the server overhead in these protocols can probably be maintained at acceptable levels. We notice that the 25 server CWD case already has almost the same client request latency as the base case.

5. Conclusions

When a server's data is uncacheable, it makes sense to replicate that server around the network so that clients will have a better chance of avoiding congestion when communicating with it. Once the server is replicated, however, the challenge becomes choosing the correct mirror for a client to use. We have investigated mechanisms for picking a server based on a client's multicast query message. These mechanisms exist purely at the application layer and require no additional network probing. We used trace-based simulations to explore two anycast methods: clustered TTL with scoping, and clustering with discard. The former used a client-initiated expanding ring search to find the nearest cluster. The latter used the hop count from the client to the server to determine the appropriate cluster for a client. In both mechanisms, once a client found a cluster, its requests would be handed off to the least-loaded member of the cluster.

The simulations show that a reasonable method of application-layer anycast can be derived from our proposed mechanisms. The time to find a server imposes another 5% of overhead on data transfer time on a lightly congested network. With

an amount of network congestion comparable to that in the Internet, we see that our anycast methods can outperform the base case. We believe that an application layer anycast service provides new ground for emerging multimedia applications, and can significantly improve the performance of the current Internet.

References

[1] BHATTACHARJEE, S. – AMMAR, E. M. – ZEGURA, E. – SHAH, V. – FEI, Z.: Application Layer Anycasting. In *Proceedings of IEEE Infocom '97*, 1997.
[2] Bright Tiger Home Page. http://www.brighttiger.com/.
[3] A Distributed Testbed for National Information Provisioning. http://ircache.nlanr.net/Cache/.
[4] FRANCIS, P.: A Call for an Internet-wide Host Proximity Service (HOPS). http://www.ingrid.org/hops/wp.html.
[5] GUTTMAN, E. – PERKINS, C. – VEIZADES, J.: Service Location Protocol. draftietf–svrloc–protocol–v2–00.txt.
[6] GUYTON, J. D. – SCHWARTZ, M. F.: Locating Nearby Copies of Replicated Internet Services. Technical Report CU–CS–762–95, University of Colorado at Boulder, 1995.
[7] LEVINE, B. N. – GARCIA–LUNA–ACEVES, J. J.: Improving Internet Multicast with Routing Labels. In *Proceedings of ICNP '97*, 1997.
[8] MATHIS, M – SEMKE, J. – MAHDAVI, J.: The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communication Review*, 27(3), July 1997.
[9] PARTRIDGE, C. – MENDEZ, T. – MILLIKEN, W.: RFC 1546: Host Anycasting Service, October 1993.